ECE 199 Final Exam Spring 2004

Friday, May 7th, 2004

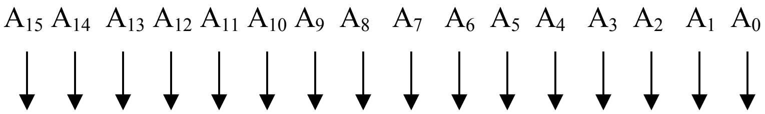
Name:

- **Be sure your exam booklet has 14 pages.**
- **Write your name at the top of each page.**
- **This is a closed book exam.**
- **You are allowed three handwritten 8.5 x 11” sheets of notes.**
- **Absolutely no interaction between students is allowed.**
- **Show all of your work.**
- **Be sure to clearly indicate any assumptions that you make.**
- **More challenging questions are marked with a *****
- **Don’t panic, and good luck!**

Problem 1	20 points	______________________
Problem 2	20 points	______________________
Problem 3	20 points	______________________
Problem 4	20 points	______________________
Problem 5	20 points	______________________
Total	100 points	

Problem 1 (20 points): Short Answer

Part A (5 points): Given a 16-bit register **A** holding a number in 2's complement form, use a single gate (e.g., AND, OR, XOR, NAND, NOR, NOT) with an arbitrary number of inputs to implement a circuit that produces an output 1 if the number is divisible by 16, and an output 0 otherwise.



Part B (5 points): You and your friends decide to collect all fifty of the U.S. state quarters. Given that each collection may or may not contain any given quarter, how many bits are necessary to represent a single collection?

Name: ____

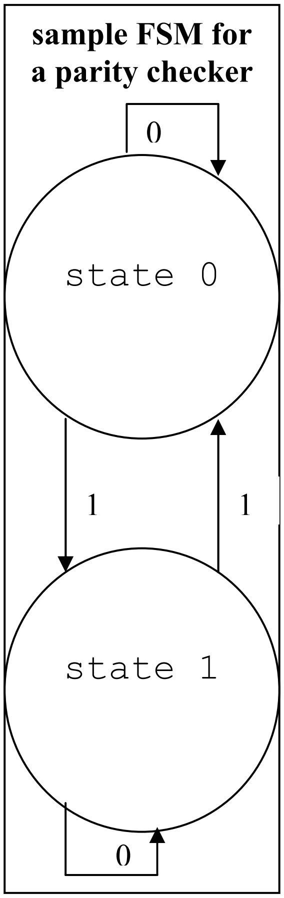
Part C (10 points): The three file I/O functions fgets, fscanf, and fread are all used to read data from a file, but the context in which each is used differs. Explain the differences between their usages, and provide an example of how each function could be used. The function signatures appear below for your convenience.

```
char*  fgets  (char* buf, int buf_size, FILE* in_file);
int    fscanf (FILE* in_file, const char* format_string, …);
size_t fread  (void* buf, size_t size, size_t n_items, FILE* in_file);
```

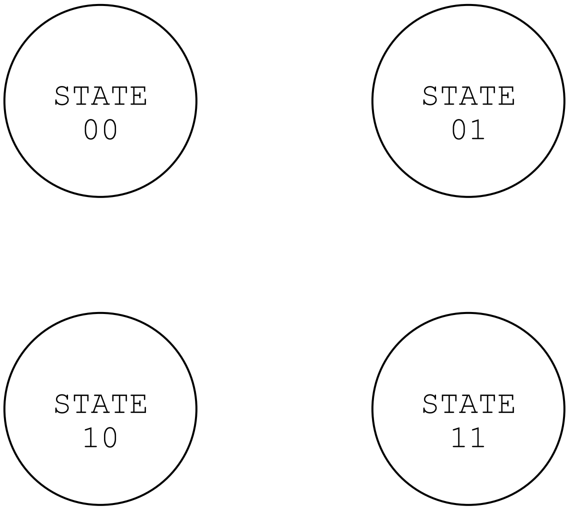
Problem 2 (20 points): LC-3 Assembly

The first two parts of the problem refer to the LC-3 code given below.

```
FSM_CHECK          LD           R1, NEG_ASCII_ZERO
                   AND          R2, R2, #0
                   ST           R2, STATE
LOOP_TOP           LDR          R3, R0, #0
                   BRz          DONE
                   LD           R2, STATE
                   ADD          R3, R3, R1
                   BRnp         HANDLE_ONE
                   ADD          R2, R2, #1
                   ADD          R3, R2, #-3
                   BRn          NEXT_CHAR
                   ADD          R2, R2, #-1
                   BRnzp        NEXT_CHAR
HANDLE_ONE         AND          R2, R2, x1E
NEXT_CHAR          ST           R2, STATE
                   ADD          R0, R0, #1
                   BRnzp        LOOP_TOP
DONE               AND          R0, R0, #0
                   LD           R2, STATE
                   BRnp         RET_ZERO
                   ADD          R0, R0, #1
RET_ZERO           RET

STATE              .BLKW        #1
NEG_ASCII_ZERO     .FILL        xFFD0
```


sample FSM for a parity checker

Part A (12 points): The LC-3 subroutine above uses a finite state machine (FSM) to check whether a string of ASCII 0's and 1's matches a particular pattern. The string (address of the first character) is passed in R0, and is assumed to contain only ASCII 0's and 1's and to be NUL-terminated. The subroutine returns R0=1 if the string matches the pattern, and R0=0 if it does not. Draw the FSM implemented by the subroutine, using the diagram below as a starting point. An example of a complete state diagram is provided to the right above.



Part B (3 points): In one sentence, describe what pattern is recognized by the code.

Part C (5 points)**:** Describe two advantages of using subroutines in assembly, and explain how the erroneous subroutine calling convention shown in the code below negates either of the two advantages that you listed.

```
            LD     R1, NUM1
            LD     R2, NUM2
            LD     R3, NUM3
            JSR    ADD3SUB
JUMPBACK    HALT

NUM1        .FILL  x0005
NUM2        .FILL  x0010
NUM3        .FILL  xFFFF

ADD3SUB     ADD    R0, R1, R2
            ADD    R0, R0, R3
            BRnzp  JUMPBACK
```

Problem 3 (20 points): Comprehending C

Part A (6 points): Write down the values of `i` and `j` after the execution of each of the following fragments of code. The three subproblems are independent.

i)

```
int i = 5;
int j = 0;

do {
      i--;
      j++;
} while ( j < 0);
```

ii)

```
int i = 1;
int j = 3;

for ( i = j ; i < j ; i++ ) {
      i++;
}
```

iii)

```
int i = 1;
int j = -6;

for( j++ ; j < 0 ; j++ ) {
      j++;
      i--;
}
```

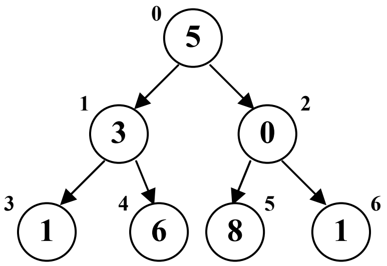
The remainder of this problem relates to a tree of integers represented as the global array of structures of type `node_t` shown below. This array is similar to the array of virtual pages in MP2. Each node has two fields that hold the array indices of its left and right children. If a node has no left or right child, the corresponding field is set to –1.

Name: ____

```
typedef struct node_t node_t;
struct node_t {
      int value;
      int left;
      int right;
}
node_t array[7];
```

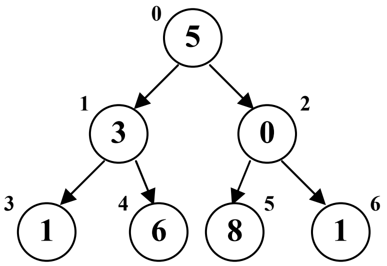
Part B (8 points): Assume that the global array holds values representing the tree shown to the right of the code. The numbers inside the circles represent the node values, while the numbers outside represent the array indices of the nodes (the root of the tree is element 0). Write the output produced by the code below when called with `i=0`, and describe in one or two sentences what the code does.

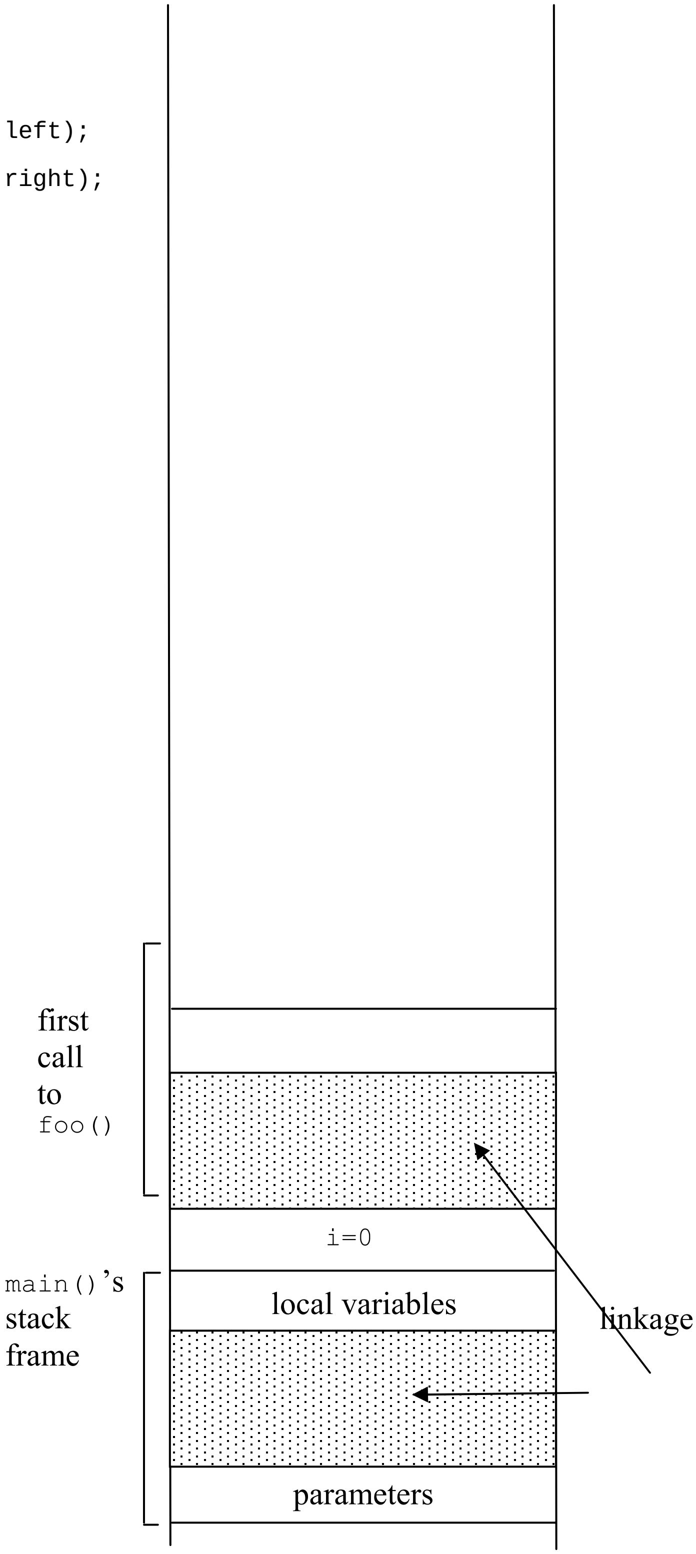
```
int foo(int i)
{
      int num = 0;

      if(array[i].left != -1)
            num = num + foo(array[i].left);
      if(array[i].right != -1)
            num = num + foo(array[i].right);
      num = num + array[i].value;

      printf("%d\n", num );

      return num;
}
```



Part C (6 points): For the function and tree used in Part B (replicated below for convenience), draw the state of the stack when the number 8 is printed to the screen, starting with the stack frame for the `foo(0)` call. The stack frame for `main()` and part of the first call to `foo()` have been drawn for you. Use the same style to draw all other stack frames for `foo()`. You need not include the stack frame for `printf()`.

```
int foo(int i)
{
      int num = 0;

      if(array[i].left != -1)
            num = num + foo(array[i].left);
      if(array[i].right != -1)
            num = num + foo(array[i].right);
      num = num + array[i].value;

      printf("%d\n", num );

      return num;
}
```



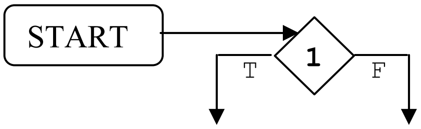


Problem 4 (20 Points): Testing and Debugging

A student has written the `blackjack_total()` function below to calculate the total of the first two cards dealt to a player and to print messages to the player The two input parameters are the count values of the cards, with Aces represented as 1.

```
      int blackjack_total(int card1, int card2)
      {
1           if (card1 + card2 > 21) {
2                 printf("You busted!\n");
3           } else if (card1 + card2 == 21) {
4                 printf("Blackjack!  You win!\n");
5           } else if (card1 + card2 < 12 && (card1==1 || card2==1)) {
6                 if (card1 + card2 == 11) {
7                       printf("Blackjack!  You win!\n");
8                 } else {
9                       printf("Want to pick another card?\n");
10                }
11                return (card1 + card2 + 10);
12          } else {
13                printf("Want to pick another card?\n");
14          }
15          return (card1 + card2);
      }
```

Part A (7 points): Draw a flowchart for the function `blackjack_total()`. Use the line numbers to the left of the code to represent conditions and statements, as shown below for the first `if` statement.



Part B (3 points): What is the minimum number of times that the `blackjack_total()` function must be called (changing the arguments each time) in order to test all paths through the function? *Hint: look at your flow chart.*

Part C (5 points): The following code is supposed to print the even numbers between one and ten, but it has a simple bug that causes the program to enter an infinite loop and not print anything to the screen. Explain the bug and correct it.

```
int main(void)
{
      int x = 1;
      while (x <= 10)
            if (!(x % 2))
                  printf("%d\n", x);
            x = x + 1;
      printf("All done!\n");
      return 0;
}
```

*****Part D** (5 points): Due to a specification ambiguity, the binary search code below sometimes returns incorrect results in the sense that the array element identified by the function does **NOT** match the `find` parameter. Explain the problem and suggest a way to fix it.

```
int binary_search (char* find, char* element[], int num_elements)
{
      int left = 0, right = num_elements - 1, middle;
      int comparison;

      while (left <= right) {
            middle = (left + right) / 2;
            comparison = strcmp (find, element[middle]);
            if (comparison == 0)
                  break;
            if (comparison < 0)
                  right = middle - 1;
            else
                  left = middle + 1;
      }
      return middle;
}
```

Problem 5 (20 points): C Structures and Pointers

You are given a list holding letters of a word in some scrambled order. Each letter of the word is stored in a structure of type `puzzle_t`.

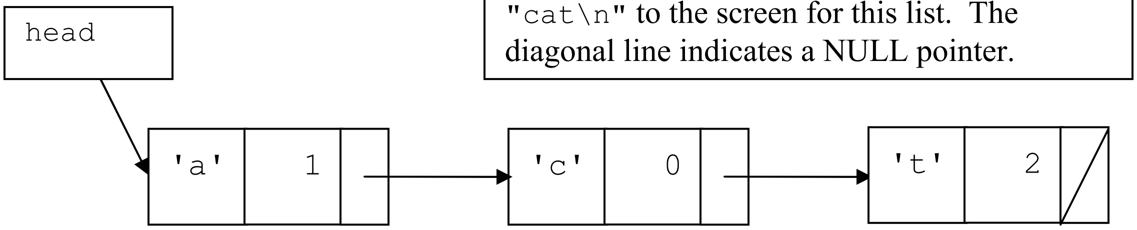
```
typedef struct puzzle_t puzzle_t;
struct puzzle_t {
      char letter;      /* the actual letter                          */
      int index;        /* the index of the letter in a string        */
      puzzle_t* next;   /* pointer to the next list element           */
}
```

Write a C function, `void solvePuzzleString(puzzle_t* start)`, that unscrambles the word held in the linked list into a dynamically allocated string, prints the string, then frees the memory used for the string. Use the template on the following page, in which the part of the code that prints and frees the dynamically allocated string has been written for you. You may assume that the all indices from 0 to N-1 appear in the list for some value of N. You may also assume that any call to a memory allocation function succeeds. An example is shown at the bottom of this page.

Here are the signatures of the standard memory allocation functions:

```
      void* malloc  (size_t size);
      void* calloc  (size_t n_elem, size_t elem_size);
      void* realloc (void* ptr, size_t size);
```

`solvePuzzleString(head)` should print `"cat\n"` to the screen for this list. The diagonal line indicates a NULL pointer.



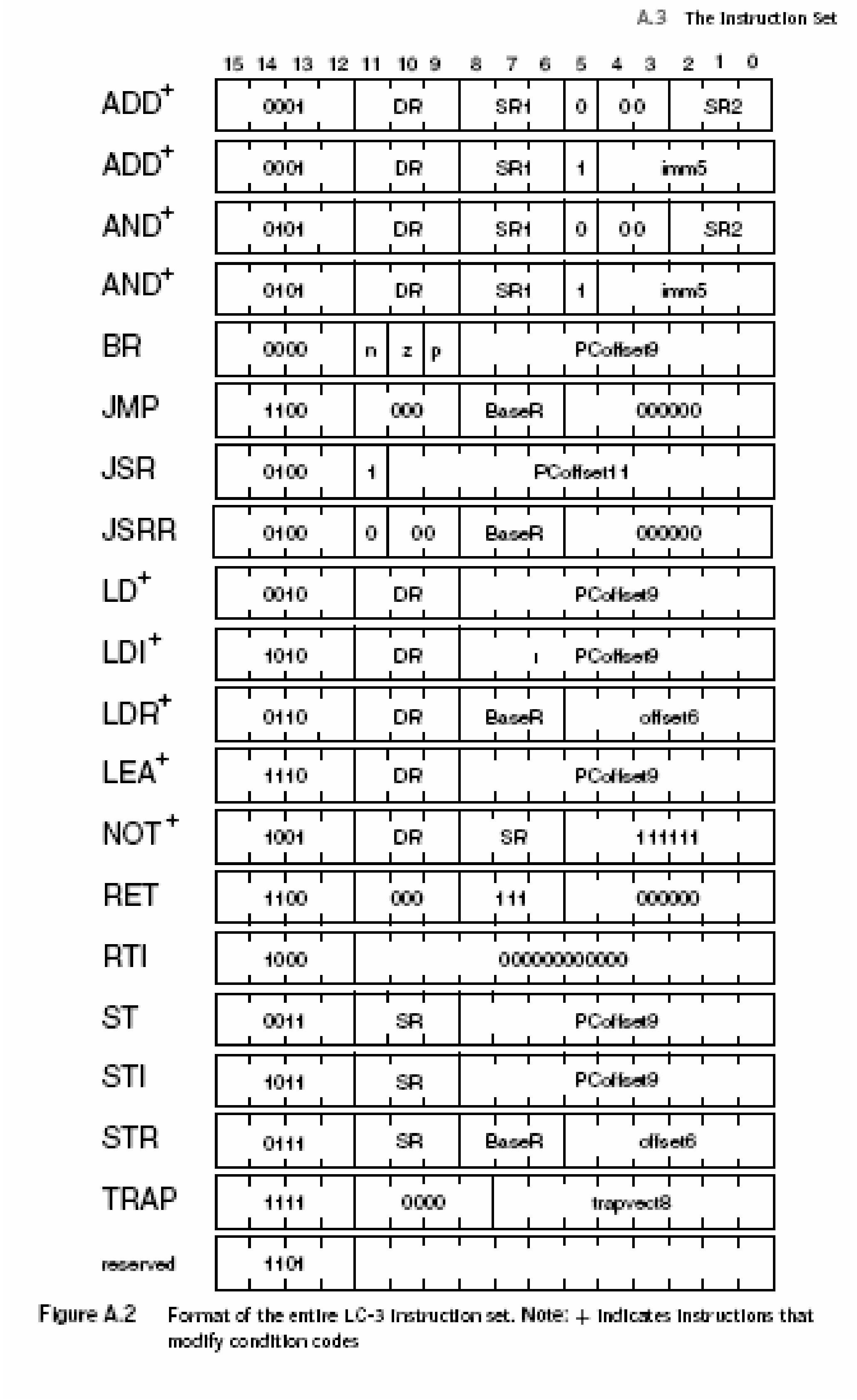
```
void solvePuzzleString(puzzle_t* start)
{
      char* stringPtr = NULL;
      /* All of your code must go here.
         Note that you CAN declare additional variables. */



      /* Print the solved puzzle string to screen */
      printf( "%s\n", stringPtr );

      /* Return memory allocated for string to heap */
      free(stringPtr);
}
```

-- End of Exam --

Name: ____

Use this page for scratchwork

0



A.3 The Instruction Set

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADD^{+}	0001				DR			SR1			0	00		SR2		
ADD^{+}	0001				DR			SR1			1	imm5				
AND^{+}	0101				DR			SR1			0	00		SR2		
AND^{+}	0101				DR			SR1			1	imm5				
BR	0000				n	z	p	PCoffset9								
JMP	1100				000			BaseR			000000					
JSR	0100				1	PCoffset11										
JSRR	0100				0	00		BaseR			000000					
LD^{+}	0010				DR			PCoffset9								
LDI^{+}	1010				DR			PCoffset9								
LDR^{+}	0110				DR			BaseR			offset6					
LEA^{+}	1110				DR			PCoffset9								
NOT^{+}	1001				DR			SR			111111					
RET	1100				000			111			000000					
RTI	1000				000000000000											
ST	0011				SR			PCoffset9								
STI	1011				SR			PCoffset9								
STR	0111				SR			BaseR			offset6					
TRAP	1111				0000				trapvect8							
reserved	1101															

Figure A.2 Format of the entire LC-3 instruction set. Note: + indicates instructions that modify condition codes